Mesh-Oriented datABase (MOAB) Version 4.0

User's Guide

Timothy J. Tautges
Jason A. Kraftcheck
Brandon M. Smith
Hong-Jun Kim

Table of Contents

List of Figures

List of Tables

1. Introduction

In scientific computing, systems of partial differential equations (PDEs) are solved on computers. One of the most widely used methods to solve PDEs numerically is to solve over discrete neighborhoods or "elements" of the domain. Popular methods include Finite Difference (FD), Finite Element (FE), and Finite Volume (FV). These methods require the decomposition of the domain into a discretized representation, which is referred to as a "mesh". The mesh is one of the fundamental types of data linking the various tools in the analysis process (mesh generation, analysis, visualization, etc.). Thus, the representation of mesh data and operations on those data play a very important role in PDE-based simulations.

MOAB is a component for representing and evaluating mesh data. MOAB can store structured and unstructured mesh, consisting of elements in the finite element "zoo", along with polygons and polyhedra. The functional interface to MOAB is simple, consisting of only four fundamental data types. This data is quite powerful, allowing the representation of most types of metadata commonly found on the mesh. MOAB is optimized for efficiency in space and time, based on access to mesh in chunks rather than through individual entities, while also versatile enough to support individual entity access.

The MOAB data model consists of the following four fundamental types: mesh interface instance, mesh entities (vertex, edge, tri, etc.), sets, and tags. Entities are addressed through handles rather than pointers, to allow the underlying representation of an entity to change without changing the handle to that entity. Sets are arbitrary groupings of mesh entities and other sets. Sets also support parent/child relationships as a relation distinct from sets containing other sets. The directed-graph provided by set parent/child relationships is useful for modeling topological relations from a geometric model and other metadata. Tags are named data which can be assigned to the mesh as a whole, individual entities, or sets. Tags are a mechanism for attaching data to individual entities and sets are a mechanism for describing relations between entities; the combination of these two mechanisms is a powerful yet simple interface for representing metadata or application-specific data. For example, sets and tags can be used together to describe geometric topology, boundary condition, and inter-processor interface groupings in a mesh.

Various mesh-related tools are provided with MOAB or can be used directly with MOAB. These tools can be used for mesh format translation, mesh skinning, solution transfer between meshes, ray tracing and other geometric searches, visualization, and relation between mesh and geometric models. These tools are described later in this document.

MOAB is written in the C++ programming language, and provides a C++ interface accessed primarily through a single MBInterface class. MOAB also implements the iMesh interface, which is specified in C but can be called directly from other languages. Almost all of the functionality in MOAB can be accessed through the iMesh interface. MOAB is developed and supported primarily on Linux and related operating systems, but can also be used on MacOS and MS Windows. MOAB can be used on parallel computing systems as well, including both clusters and high-end parallel systems like IBM BG/P and Cray systems. MOAB is released under a standard LGPL open source software license.

MOAB is used in several ways in various applications. MOAB serves as the underlying mesh data representation in several scientific computing applications [1]. MOAB can also be used as a mesh format translator, using readers and writers included in MOAB. MOAB has also been used as a bridge to couple results in multi-physics analysis and to link these applications with other mesh services [2].

The remainder of this report is organized as follows. Section 2, "Getting Started", provides a few simple examples of using MOAB to perform simple tasks on a mesh. Section 3 discusses the MOAB data model in more detail, including some aspects of the implementation. Section 4 summarizes the MOAB function API. Section 5 describes some of the tools included with MOAB, and the implementation of mesh readers/writers for MOAB. Section 6 contains a brief description of MOAB's relation to the TSTT mesh interface. Section 7 gives a conclusion and future plans for MOAB development. Section 8 gives references cited in this report. A reference description of the full MOAB API is contained in Section 9.

Several other sources of information about MOAB may also be of interest to readers. Meta-data conventions define how sets and /or tags are used together to represent various commonly-used simulation constructs;

conventions used by MOAB are described in Ref [4]. This document is maintained separately from this document, since it is expected to change over time. The MOAB project maintains a wiki [5], which links to most MOAB-related information. MOAB also uses several mailing lists [6],[7] for MOAB-related discussions. Potential users are encouraged to interact with the MOAB team using these mailing lists.

2. MOAB Data Model

The MOAB data model describes the basic types used in MOAB and the language used to communicate that data to applications. This chapter describes that data model, along with some of the reasons for some of the design choices in MOAB.

2.1. MOAB Interface

MOAB is written in C++. The primary interface with applications is through member functions of the abstract base class MBInterface. The MOAB library is created by instantiating MBCore, which implements the MBInterface API. Multiple instances of MOAB can exist concurrently in the same application; mesh entities are not shared between these instances[1]. MOAB is most easily viewed as a database of mesh objects accessed through the instance. No other assumptions explicitly made about the nature of the mesh stored there; for example, there is no fundamental requirement that elements fill space or do not overlap each other geometrically.

2.2. Mesh Entities

MOAB represents the following topological mesh entities: vertex, edge, triangle, quadrilateral, polygon, tetrahedron, pyramid, prism, knife, hexahedron, polyhedron. MOAB uses the MBEntityType enumeration to refer to these entity types (see Table 1). This enumeration has several special characteristics, chosen intentionally: the types begin with vertex, entity types are grouped by topological dimension, with lower-dimensional entities appearing before higher dimensions; the enumeration includes an entity type for sets (described in the next section); and MBMAXTYPE is included at the end of this enumeration, and can be used to terminate loops over type. In addition to these defined values, the an increment operator (++) is defined such that variables of type MBEntityType can be used as iterators in loops.

MOAB refers to entities using "handles". Handles are implemented as long integer data types, with the four highest-order bits used to store the entity type (mesh vertex, edge, tri, etc.) and the remaining bits storing the entity id. This scheme is convenient for applications because:

- Handles sort lexicographically by type and dimension; this can be useful for grouping and iterating over entities by type.
- The type of an entity is indicated by the handle itself, without needing to call a function.
- Entities allocated in sequence will typically have contiguous handles; this characteristic can be used to efficiently store and operate on large lists of handles.

This handle implementation is exposed to applications intentionally, because of optimizations that it enables, and is unlikely to change in future versions.

Table 1: Values defined for the MBEntityType enumerated type.

MBVERTEX = 0		MBPRISM
MBEDGE		MBKNIFE
MBTRI		MBHEX
MBQUAD		MBPOLYHEDRON
MBPOLYGON		MBENTITYSET
MBTET		MBMAXTYPE

[1] One exception to this statement is when the parallel interface to MOAB is used; in this case, entity sharing between instances is handled explicitly using message passing. This is described in more detail in Section 5 of this document.

MBPYRAMID		

MOAB defines a special class for storing lists of entity handles, named MBRange. This class stores handles as a series of (start_handle, end_handle) subrange tuples. If a list of handles has large contiguous ranges, MBRange provides almost constant-size storage for these lists. Since entities are typically created in groups, e.g. during mesh generation or file import, a high degree of contiguity in handle space is typical. MBRange provides an interface similar to C++ STL containers like vector, containing iterator data types and functions for initializing and iterating over entity handles stored in the range, MBRange also provides functions for efficient Boolean operations like subtraction and intersection. Most API functions in MOAB come in both range-based and vector-based variants. By definition, a list of entities stored in an MBRange is always sorted, and can contain a given entity handle only once.

Typical usage of an MBRange object would look like:

```
int my_function(MBRange &from_range) {
  int num_in_range = from_range.size();
  MBRange to_range;
  MBRange::iterator rit;
 for (rit = from_range.begin(); rit < from_range.end(); rit++) {
    MBEntityHandle this_ent = *rit;
    to_range.insert(this_ent);
  }
}
```

Here, the range is iterated over similar to how an STL vector would be iterated over.

2.2.1. Adjacencies & AEntities

The term adjacencies is used to refer to those entities topologically connected to a given entity, e.g. the faces bounded by a given edge or the vertices bounding a given region. MOAB provides functions for querying adjacent entities by target dimension, using the same functions for higher- and lower-dimension adjacencies. By default, MOAB stores the minimum data necessary to recover adjacencies between entities. When a mesh is initially loaded into MOAB, only entity-vertex (i.e. “downward”) adjacencies are stored, in the form of entity connectivity. When “upward” adjacencies are requested for the first time, e.g. from vertices to regions, MOAB stores all vertex-entity adjacencies explicitly, for all entities in the mesh. Non-vertex entity to entity adjacencies are never stored, unless explicitly requested by the application.

In its most fundamental form, a mesh need only be represented by its vertices and the entities of maximal topological dimension. For example, a hexahedral mesh can be represented as the connectivity of the hex elements and the vertices forming the hexes. Edges and faces in a 3D mesh need not be explicitly represented. We refer to such entities as “AEntities”, where ‘A’ refers to “Auxiliary”, “Ancillary”, and a number of other words mostly beginning with ‘A’. Individual AEntities are created only when requested by applications, either using mesh modification functions or by requesting adjacencies with a special “create if missing” flag passed as “true”. This reduces the overall memory usage when representing large meshes. Note entities must be explicitly represented before they can be assigned tag values or added to entity sets (described in following Sections).

2.3. Entity Sets

Entity sets are used to store arbitrary collections of entities and other sets. Sets are used for a variety of things in mesh-based applications, from the set of entities discretizing a given geometric model entity to the entities partitioned to a specific processor in a parallel finite element application. MOAB entity sets can also store parent/child relations with other entity sets, with these relations distinct from contains relations. Parent/child relations are useful for building directed graphs with graph nodes representing collections of mesh entities; this construct can be used, for example, to represent an interface of mesh faces shared by two distinct collections of mesh regions. MOAB also defines one special set, the “root set” or the interface itself; all entities are part of this

set by definition. Defining a root set allows the use of a single set of MOAB API functions to query entities in the overall mesh as well as its subsets.

MOAB entity sets can be one of two distinct types: list-type entity sets preserve the order in which entities are added to the set, and can store a given entity handle multiple times in the same set; set-type sets are always ordered by handle, regardless of the order of addition to the set, and can store a given entity handle only once. This characteristic is assigned when the set is created, and cannot be changed during the set's lifetime.

MOAB provides the option to track or not track entities in a set. When entities (and sets) are deleted by other operations in MOAB, they will also be removed from containing sets for which tracking has been enabled. This behavior is assigned when the set is created, and cannot be changed during the set's lifetime. The cost of turning tracking on for a given set is sizeof(MBEntityHandle) for each entity added to the set; MOAB stores containing sets in the same list which stores adjacencies to other entities.

Using an entity set looks like the following:

```
// load a file using MOAB, putting the loaded mesh into a file set
MBEntityHandle file_set;
MBErrorCode rval = moab->load_file("fname.vtk", …, file_set);
MBRange set_ents;
// get all the 3D entities in the set
rval = moab->get_entities_by_dimension(file_set, 3, set_ents);
```

Entity sets are often used in conjunction with tags (described in the next section), and provide a powerful mechanism to store a variety of meta-data with meshes.

2.4. Tags

Applications of a mesh database often need to attach data to mesh entities. The types of attached data are often not known at compile time, and can vary across individual entities and entity types. MOAB refers to this attached data as a "tag". Tags can be thought of loosely as a variable, which can be given a distinct value for individual entities, entity sets, or for the interface itself. A tag is referenced using a handle, similarly to how entities are referenced in MOAB. Each MOAB tag has the following characteristics, which can be queried through the MOAB interface:

- Name
- Size (in bytes)
- Storage type
- Data type (integer, double, opaque, entity handle)
- Handle

The storage type determines how tag values are stored on entities.

- Dense: Dense tag values are stored in arrays which match arrays of contiguous entity handles. Dense tags are more efficient in both storage and memory if large numbers of entities are assigned the same tag. Storage for a given dense tag is not allocated until a tag value is set on an entity; memory for a given dense tag is allocated for all entities in a given sequence at the same time.
- Sparse: Sparse tags are stored as a list of (entity handle, tag value) tuples, one list per sparse tag, sorted by entity handle.
- Bit: Bit tags are stored similarly to dense tags, but with special handling to allow allocation in bit-size amounts per entity.

MOAB also supports variable-length tags, which can have a different length for each entity they are assigned to. Variable length tags are stored similarly to sparse tags.

The data type of a tag can either be one understood at compile time (integer, double, entity handle), in which case the tag value can be saved and restored properly to/from files and between computers of different architecture (MOAB provides a native HDF5-based save/restore format for this purpose; see Section 4.6). The opaque data type is used for character strings, or for allocating "raw memory" for use by applications (e.g. for storage application-defined structures or other abstract data types). These tags are saved and restored as raw memory, with no special handling for endian or precision differences.

An application would use the following code to attach a double-precision tag to vertices in a mesh, e.g. to assign a temperature field to those vertices:

```
// load a file using MOAB and get the vertices
MBErrorCode rval = moab->load_file("fname.vtk");
MBRange verts;
rval = moab->get_entities_by_dimension(file_set, 0, verts);
// create a tag called "TEMPERATURE"
MBTag temperature;
double def_val = -1.0d-300, new_val = 273.0;
rval = moab->tag_create("TEMPERATURE", sizeof(double), MB_TAG_DENSE,
                        MB_TYPE_DOUBLE, temperature, &def_val);
// assign a value to vertices
for (MBRange::iterator vit = verts.begin();
     vit != verts.end(); vit++)
  rval = moab->tag_set_data(temperature, &(*rit), 1, &new_val);
```

The semantic meaning of a tag is determined by applications using it. However, to promote interoperability between applications, there are a number of tag names reserved by MOAB which are intended to be used by convention. At this time, MOAB defines the tags in Error: Reference source not found as having conventional semantics. Mesh readers and writers in MOAB use these tag conventions, and applications can use them as well to access the same data. Ref. [4] maintains an up-to-date list of conventions for meta-data usage in MOAB.

3. MOAB API Design Philosophy and Summary

This section describes the design philosophy behind MOAB, and summarizes the functions, data types and enumerated variables in the MOAB API. A complete description of the MOAB API is available in online documentation in the MOAB distribution [8].

MOAB is designed to operate efficiently on collections of entities. Entities are often created or referenced in groups (e.g. the mesh faces discretizing a given geometric face, the 3D elements read from a file), with those groups having some form of temporal or spatial locality. The interface provides special mechanisms for reading data directly into the native storage used in MOAB, and for writing large collections of entities directly from that storage, to avoid data copies. MOAB applications structured to take advantage of that locality will typically operate more efficiently.

MOAB has been designed to maximize the flexibility of mesh data which can be represented. There is no explicit constraint on the geometric structure of meshes represented in MOAB, or on the connectivity between elements. In particular, MOAB allows the representation of multiple entities with the same exact connectivity; however, in these cases, explicit adjacencies must be used to distinguish adjacencies with AEntities bounding such entities.

The number of vertices used to represent a given topological entity can vary, depending on analysis needs; this is often the case in FEA. For example, applications often use "quadratic" or 10-vertex tetrahedral, with vertices at edge midpoints as well as corners. MOAB does not distinguish these variants by entity type, refering to all variants as "tetrahedra". The number of vertices for a given entity is used to distinguish the variants, with canonical numbering conventions used to determine placement of the vertices [9]. This is similar to how such variations are represented in the Exodus [10] and Patran [11] file formats. In practice, we find that this simplifies coding in applications, since in many cases the handling of entities depends only on the number of corner vertices

in the element. Some MOAB API functions provide a flag which determines whether corner or all vertices are requested.

The MOAB API is designed to balance complexity and ease of use. This balance is evident in the following general design characteristics:

- **Entity lists:** Lists of entities are passed to and from MOAB in a variety of forms. Lists output from MOAB are passed as either STL vector or MBRange data types. Either of these constructs may be more efficient in both time and memory, depending on the semantics of the data being requested. Input lists are passed as either MBRange's, or as a pointer to MBEntityHandle and a size. The latter allows the same function to be used when passing individual entities, without requiring construction of an otherwise unneeded STL vector.
- **Entity sets:** Most query functions accept an entity set as input. Applications can pass zero to indicate a request for the whole interface. Note that this convention applies only to query functions; attempts to add or subtract entities to/from the interface using set-based modification functions, or to add parents or children to the interface set, will fail. Allowing specification of the interface set in this manner avoids the need for a separate set of API functions to query the database as a whole.
- **Implicit Booleans in output lists:** A number of query functions in MOAB allow specification of a Boolean operation (MBInterface::INTERSECT or MBInterface::UNION). This operation is applied to the results of the query, often eliminating the need for code the application would need to otherwise implement. For example, to find the set of vertices shared by a collection of quadrilaterals, the application would pass that list of quadrilaterals to a request for vertex adjacencies, with MBInterface::INTERSECT passed for the Boolean flag. The list of vertices returned would be the same as if the application called that function for each individual entity, and computed the intersection of the results over all the quadrilaterals. Applications may also input non-empty lists to store the results, in which case the intersection is also performed with entities already in the list. In many cases, this allows optimizations in both time and memory inside the MOAB implementation.

Since these objectives are at odds with each other, tradeoffs had to be made between them. Some specific issues that came up are:

- **Using ranges:** Where possible, entities can be referenced using either ranges (which allow efficient storage of long lists) or STL vectors (which allow list order to be preserved), in both input and output arguments.
- **Entities in sets:** Accessing the entities in a set is done using the same functions which access entities in the entire mesh. The whole mesh is referenced by specifying a set handle of zero[2].
- **Entity vectors on input:** Functions which could normally take a single entity as input are specified to take a vector of handles instead. Single entities are specified by taking the address of that entity handle and specifying a list length of one. This minimizes the number of functions, while preserving the ability to input single entities.[3]

Table 2 lists basic data types and enumerated variables defined and used by MOAB. Values of the MBErrorCode enumeration are returned from most MOAB functions, and can be compared to those listed in Appendix [ref-appendix].

MOAB uses several pre-defined tag names to define data commonly found in various mesh-based analyses. Ref. [4] describes these meta-data conventions in more detail. These conventions will be added to as new conventions emerge for using sets and tags in MOAB applications.

[2] In iMesh, the whole mesh is specified by a special entity set handle, referred to as the "root set".
[3] Note that STL vectors of entity handles can be input in this manner by using &vector[0] and vector.size() for the 1d vector address and size, respectively.

Table 2: Basic data types and enums defined in MOAB.

Enum / Type	Description
MBErrorCode	Specific error codes returned from MOAB
MBEntityHandle	Type used to represent entity handles
MBTag	Type used to represent tag handles
MBTagType	Type used to represent tag storage type
MBDataType	Type used to represent tag data type

Table 3 lists the various groups of functions that comprise the MOAB API. This is listed here strictly as a reference to the various types of functionality supported by MOAB; for a more detailed description of the scope and syntax of the MOAB API, see the online documentation [8].

Table 3: Groups of functions in MOAB API. See Ref. [8] for more details.

Function group	Examples	Description
Constructor, destructor, interface	MBInterface, ~MBCore, query_interface	Construct/destroy interface; get pointer to read/write interface
Entity query	get_entities_by_dimension, get_entities_by_handle	Get entities by dimension, type, etc.
Adjacencies	get_adjacencies, set_adjacencies, add_adjacencies	Get topologically adjacent entities; set or add explicit adjacencies
Vertex coordinates	get_coords, set_coords	Get/set vertex coordinates
Connectivity	get_connectivity, set_connectivity	Get/set connectivity of non-vertex entities
Sets	create_meshset, add_entities, add_parent_child	Create and work with entity sets
Tags	tag_get_data, tag_create	Create, read, write tag data
Handles	type_from_handle, id_from_handle	Go between handles and types/ids
File handling	load_mesh, save_mesh	Read/write mesh files
Geometric dimension	get_dimension, set_dimension	Get/set geometric dimension of mesh
Mesh modification	create_vertex, delete_entity	Create or delete mesh entities
Information	list_entities, get_last_error	Get or print certain information
High-order nodes	high_order_node	Get information on high-order nodes
Canonical numbering	side_number	Get canonical numbering information

4. Related Mesh Services

A number of mesh-based services are often used in conjunction with a mesh library. For example, parallel applications often need to visualize the mesh and associated data. Other services, like spatial interpolation or

finding the faces on the "skin" of a 3D mesh, can be implemented more efficiently using knowledge of specific data structures in MOAB. Several of these services provided with MOAB are described in this chapter.

4.1. Visualization

Visualization is one of the most common needs associated with meshes. The primary tool used to visualize MOAB meshes is VisIt [12]. Users can specify that VisIt read mesh directly out of the MOAB instance, by specifying the ITAPS-MOABC mesh format and a file readable by MOAB (see xxx).

There are some initial capabilities in VisIt for limited viewing and manipulation of tag data and some types of entity sets. Tag data is visualized using the same mechanisms used to view other field data in VisIt, e.g. using a pseudocolor plot; sets are viewed using VisIt's SIL window, accessed by selecting the SIL icon in the data selection window. xxx shows a vertex-based radiation temperature field computed by the Cooper rad-hydro code [1] for a subset of geometric volumes in a mesh.

Reorganization of VisIt's set handling is also underway, to increase versatility and flexibility of this important mechanism.

4.2. Parallel Decomposition

To support parallel simulation, applications often need to partition a mesh into parts, designed to balance the load and minimize communication between sets. MOAB includes the MBZoltan tool for this purpose, constructed on the well-known Zoltan partitioning library [13]. After computing the partition using Zoltan, MBZoltan stores the partition as either tags on individual entities in the partition, or as tagged sets, one set per part. Since a partition often exhibits locality similar to how the entities were created, storing it as sets (based on MBRange's) is often more memory-efficient than an entity tag-based representation. Xxx shows a partition computed with MBZoltan (and visualized in VisIt).

4.3. Skinner

An operation commonly applied to mesh is to compute the outermost "skin" bounding a contiguous block of elements. This skin consists of elements of one fewer topological dimension, arranged in one or more topological balls on the boundary of the elements. The MBSkinner tool computes the skin of a mesh in a memory-efficient manner. MBSkinner uses knowledge about whether vertex-entity adjacencies and AEntities exist to minimize memory requirements and searching time required during the skinning process. This skin can be provided as a single collection of entities, or as sets of entities distinguished by forward and reverse orientation with respect to higher-dimensional entities in the set being skinned.

The following code fragment shows how MBSkinner can be used to compute the skin of a range of hex elements:

```
MBRange hexes, faces;
MBErrorCode rval = moab->get_entities_by_dimension(0, 3, hexes);
MBSkinner myskinner(moab);
bool verts_too = false;
MBErrorCode rval = myskinner.find_skin(hexes, verts_too, faces);
```

MBSkinner can also skin a mesh based on geometric topology groupings imported with the mesh. The geometric topology groupings contain information about the mesh "owned" by each of the entities in the geometric model, e.g. the model vertices, edges, etc. Links between the mesh sets corresponding to those entities can be inferred directly from the mesh. Skinning a mesh this way will typically be much faster than doing so on the actual mesh elements, because there is no need to create and destroy interior faces on the mesh.

4.4. Tree Decompositions

MOAB provides several mechanisms for spatial decomposition and searching in a mesh:

- MBAdaptiveKDTree: Adaptive KD tree, a space-filling decomposition with axis-aligned splitting planes, enabling fast searching.

- MBBSPTree: Binary Space Partition tree, with non-axis-aligned partitions, for fast spatial searches with slightly better memory efficiency than KD trees.
- MBOrientedBoxTreeTool: Oriented Bounding Box tree hierarchy, useful for fast ray-tracing on collections of mesh facets.

These trees have various space and time searching efficiencies. All are implemented based on entity sets and parent/child relations between those sets, allowing storage of a tree decomposition using MOAB's native file storage mechanism (see Section 4.6.1). MOAB's entity set implementation is specialized for memory efficiency when representing binary trees. Tree decompositions in MOAB have been used to implement fast ray tracing to support radiation transport [14], solution coupling between meshes [2], and embedded boundary mesh generation [15]. MOAB also includes the DAGMC tool, supporting Monte Carlo radiation transport.

The following code fragment shows very basic use of MBAdaptiveKDTree. A range of entities is put in the tree; the leaf containing a given point is found, and the entities in that leaf are returned.

```
// create the adaptive kd tree from a range of tets
MBEntityHandle tree_root
MBAdaptiveKDTree myTree(moab);
MBErrorCode rval = myTree.build_tree(tets, tree_root);

// get the overall bounding box corners
double boxmax[3], boxmin;
rval = myTree.get_tree_box(tree_root, boxmax, boxmin);

// get the tree leaf containing point xyz, and the tets in that leaf
MBAdaptiveKDTreeIter treeiter;
rval = myTree.leaf_containing_point(tree_root, xyz, treeiter);
MBRange leaf_tets;
rval = moab->get_entities_by_dimension(treeiter.handle(), 3,
                                       leaf_tets, false);
```

More detailed examples of using the various tree decompositions in MOAB can be found in [ref-treeexamples].

4.5. File Reader/Writer Interfaces

Mesh readers and writers communicate mesh into/out of MOAB from/to disk files. Reading a mesh often involves importing large sets of data, for example coordinates of all the nodes in the mesh. Normally, this process would involve reading data from the file into a temporary data buffer, then copying data from there into its destination in MOAB. To avoid the expense of copying data, MOAB has implemented a reader/writer interface that provides direct access to blocks of memory used to represent mesh.

The reader interface, declared in MBReadUtilIface, is used to request blocks of memory for storing coordinate positions and element connectivity. The pointers returned from these functions point to the actual memory used to represent those data in MOAB. Once data is written to that memory, no further copying is done. This not only saves time, but it also eliminates the need to allocate a large memory buffer for intermediate storage of these data. The reader interface consists of the following functions:

- **get_node_arrays:** Given the number of vertices requested, the number of geometric dimensions, and a requested start id, allocates a block of vertex handles and returns pointers to coordinate arrays in memory, along with the actual start handle for that block of vertices.
- **get_element_array:** Given the number of elements requested, the number of vertices per element, the element type and the requested start id, allocates the block of elements, and returns a pointer to the connectivity array for those elements and the actual start handle for that block. The number of vertices per element is necessary because those elements may include higher-order nodes, and MOAB stores these as part of the normal connectivity array.

- **update_adjacencies:** This function takes the start handle for a block of elements and the connectivity of those elements, and updates adjacencies for those elements. Which adjacencies are updated depends on the options set in AEntityFactory.

The following code fragment illustrates the use of MBReadUtilIface to read a mesh directly into MOAB's native representation. This code assumes that connectivity is specified in terms of vertex indices, with vertex indices starting from 1.

```
// get the read iface from moab
MBReadUtilIface *iface;
MBErrorCode rval = moab->get_interface("MBReadUtilIface", &iface);

// allocate a block of vertex handles and read xyz's into them
std::vector<double*> arrays;
MBEntityHandle startv, *starth;
rval = iface->get_node_arrays(3, num_nodes, 0, startv, arrays);
for (int i = 0; i < num_nodes; i++)
  infile >> arrays[0][i] >> arrays[1][i] >> arrays[2][i];

// allocate block of hex handles and read connectivity into them
rval = iface->get_element_array(num_hexes, 8, MBHEX, 0, starth);
for (int i = 0; i < 8*num_hexes; i++)
  infile >> starth[i];

// change connectivity indices to vertex handles
for (int i = 0; i < 8*num_hexes; i++)
  starth[i] += startv-1;
```

The writer interface, declared in MBWriteUtilIface, provides functions that support writing vertex coordinates and element connectivity to storage locations input by the application. Assembling these data is a common task for writing mesh, and can be non-trivial when exporting only subsets of a mesh. The writer interface declares the following functions:

- **get_node_arrays:** Given already-allocated memory and the number of vertices and dimensions, and a range of vertices, this function writes vertex coordinates to that memory. If a tag is input, that tag is also written with integer vertex ids, starting with 1, corresponding to the order the vertices appear in that sequence (these ids are used to write the connectivity array in the form of vertex indices).
- **get_element_array:** Given a range of elements and the tag holding vertex ids, and a pointer to memory, the connectivity of the specified elements are written to that memory, in terms of the indices referenced by the specified tag. Again, the number of vertices per element is input, to allow the direct output of higher-order vertices.
- **gather_nodes_from_elements:** Given a range of elements, this function returns the range of vertices used by those elements. If a bit-type tag is input, vertices returned are also marked with 0x1 using that tag. If no tag is input, the implementation of this function uses its own bit tag for marking, to avoid using an n^2 algorithm for gathering vertices.
- **reorder:** Given a permutation vector, this function reorders the connectivity for entities with specified type and number of vertices per entity to match that permutation. This function is needed for writing connectivity into numbering systems other than that used internally in MOAB.

The following code fragment shows how to use MBWriteUtilIface to write the vertex coordinates and connectivity indices for a subset of entities.

```
// get the write iface from moab
```

```
MBWriteUtilIface *iface;
MBErrorCode rval = moab->get_interface("MBWriteUtilIface", &iface);

// get all hexes the model, and choose the first 10 of those
MBRange tmp_hexes, hexes, verts;
rval = moab->get_entities_by_type(0, MBHEX, tmp_hexes);
for (int i = 0; i < 10; i++) hexes.insert(tmp_hexes[i]);
rval = iface->gather_nodes_from_elements(hexes, 0, verts);

// assign vertex ids
iface->assign_ids(verts, 0, 1);

// allocate space for coordinates & write them
std::vector<double*> arrays(3);
for (int i = 0; i < 3; i++) arrays[i] = new double[verts.size()];
iface->get_node_arrays(3, verts.size(), verts, 0, 1, arrays);

// put connect'y in array, in the form of indices into vertex array
std::vector<int> conn(8*hexes.size());
iface->get_element_array(hexes.size(), 8, 0, hexes, 0, 1, &conn[0]);
```

4.6. File Readers/Writers Packaged With MOAB

MOAB has been designed to efficiently represent data and metadata commonly found in finite element mesh files. Readers and writers are included with MOAB which import/export specific types of metadata in terms of MOAB sets and tags, as described earlier in this document. The number of readers and writers in MOAB will probably grow over time, and so they are not enumerated here. See the src/io/README file in the MOAB source distribution for a current list of supported formats.

Because of its generic support for readers and writers, described in the previous section, MOAB is also a good environment for constructing new mesh readers and writers. The ReadTemplate and WriteTemplate classes in src/io are useful starting points for constructing new file readers/writers; applications are encouraged to submit their own readers/writers for inclusion in MOAB's contrib/io directory in the MOAB source.

The usefulness of a file reader/writer is determined not only by its ability to read and write nodes and elements, but also in its ability to store the various types of meta-data included with the typical mesh. MOAB readers and writers are distinguished by their ability to preserve meta-data in meshes that they read and write. For example, MOAB's CUB reader imports not only the mesh saved from CUBIT, but also the grouping of mesh entities into sets which reflect the geometric topology of the model used to generate the mesh. See [4] for a more detailed description of meta-data conventions used in MOAB's file readers and writers, and the individual file reader/writer header files in src/io for details about the specific readers and writers.

Three specific file readers in MOAB bear further discussion: MOAB's native HDF5-based file reader/writer; the CUB reader, used to import mesh and meta-data represented in CUBIT; and the CGM reader, which imports geometric models. These are described next.

4.6.1. Native HD5-Based Reader/Writer

A mesh database must be able to save and restore the data stored in its data model, at least to the extent to which it understands the semantics of that data. MOAB defines an HDF5-based file format that can store data embedded in MOAB. By convention, these files are given an ".h5m" file extension. When reading or writing large amounts of data, it is recommended to use this file format, as it is the most complete and also the most efficient of the file readers/writers in MOAB.

4.6.2. CUB Reader

CUBIT is a toolkit for generating tetrahedral and hexahedral finite element meshes from solid model geometry [16]. This tool saves and restores data in a custom ".cub" file, which stores both mesh and geometry (and data relating the two). The CUB reader in MOAB can import and interpret much of the meta-data information saved in .cub files. Ref. [4] describes the conventions used to store this meta-data in the MOAB data model. The information read from .cub files, and stored in the MOAB data model, includes:

- Geometric model entities and topology
- Model entity names and ids
- Groups, element blocks, nodesets, and sidesets, including model entities stored in them
- Mesh scheme and interval size information assigned to model entities

Note that although information about model entities is recovered, MOAB by default does not depend on a solid modeling engine; this information is stored in the form of entity sets and parent/child relations between them. See Ref. [4] for more information.

4.6.3. CGM Reader

The Common Geometry Module (CGM) [17] is a library for representing solid model and other types of solid geometry data. The CUBIT mesh generation toolkit uses CGM for its geometric modeling support, and CGM can restore geometric models in the exact state in which they were represented in CUBIT. MOAB contains a CGM reader, which can be enabled with a configure option. Using this reader, MOAB can read geometric models, and represent their model topology using entity sets linked by parent/child relations. The mesh in these models comes directly from the modeling engine faceting routines; these are the same facets used to visualize solid models in other graphics engines. When used in conjunction with the VisIt visualization tool (see Section 4.1), this provides a solution for visualizing geometric models. Xxx shows a model imported using MOAB's CGM reader and visualized with VisIt.

5. Parallel Mesh Representation and Query

A parallel mesh representation must strike a careful balance between providing an interface to mesh similar to that of a serial mesh, while also allowing the discovery of parallel aspects of the mesh and performance of parallel mesh-based operations efficiently. MOAB supports a spatial domain-decomposed view of a parallel mesh, where each subdomain is assigned to a processor, lower-dimensional entities on interfaces between subdomains are shared between processors, and ghost entities can be exchanged with neighboring processors. Locally, each subdomain, along with any locally-represented ghost entities, are accessed through a local MOAB instance. Parallel aspects of the mesh, e.g. whether entities are shared, on an interface, or ghost entities, are embedded in the same data model (entities, sets, tags, interface) used in the rest of MOAB. MOAB provides a suite of parallel functions for initializing and communicating with a parallel mesh, along with functions to query the parallel aspects of the mesh.

5.1. Nomenclature & Representation

Before discussing how to access parallel aspects of a mesh, several terms need to be defined:

Shared entity: An entity shared by one or several other processors.

Multi-shared entity: An entity shared by more than two processors.

Owning Processor: Each shared entity is owned by exactly one processor. This processor has the right to set tag values on the entity and have those values propagated to any sharing processors.

Part: The collection of entities assigned to a given processor. When reading mesh in parallel, the entities in a Part, along with any lower-dimensional entities adjacent to those, will be read onto the assigned processor.

Partition: A collection of Parts which take part in parallel collective communication, usually associated with an MPI communicator.

Interface: A collection of mesh entities bounding entities in multiple parts. Interface entities are owned by a single processor, but are represented on all parts/processors they bound.

Ghost entity: A shared, non-interface, non-owned entity.

Parallel status: A characteristic of entities and sets represented in parallel. The parallel status, or "pstatus", is represented by a bit field stored in an unsigned character, with bit values as described in Table 4.

Table 4: Bits representing various parallel characteristics of a mesh. Also listed are enumerated values that can be used in bitmask expressions; these enumerated variables are declared in MBParallelConventions.h.

Bit	Name	Represents
0x1	PSTATUS_NOT_OWNED	Not owned by the local processor
0x2	PSTATUS_SHARED	Shared by exactly two processors
0x4	PSTATUS_MULTISHARED	Shared by three or more processors
0x8	PSTATUS_INTERFACE	Part of lower-dimensional interface shared by multiple processors
0x10	PSTATUS_GHOST	Non-owned, non-interface entities represented locally

Parallel functionality is described in the following sections. First, methods to load a mesh into a parallel representation are described; next, functions for accessing parallel aspects of a mesh are described; functions for communicating mesh and tag data are described.

5.2. Parallel Mesh Initialization

Parallel mesh is initialized in MOAB in several steps:

1. Establish a local mesh on each processor, either by reading the mesh into that representation from disk, or by creating mesh locally through the normal MOAB interface.
2. Find vertices, then other entities, shared between processors, based on a globally-consistent vertex numbering stored on the GLOBAL_ID tag.
3. Exchange ghost or halo elements within a certain depth of processor interfaces with neighboring processors.

These steps can be executed by a single call to MOAB's load_file function, using the procedure described in Section 5.2.1. Or, they can be executed in smaller increments calling specific functions in MOAB's ParallelComm class, as described in Section 5.2.2. Closely related to the latter method is the handling of communicators, described in more detail in Section.

5.2.1. Parallel Mesh Initialization by Loading a File

In the file reading approach, a mesh must contain some definition of the partition (the assignment of mesh, usually regions, to processors). Partitions can be derived from other set structures already on the mesh, or can be computed explicitly for that purpose by tools like mbzoltan (see Section 4.2). For example, geometric volumes used to generate the mesh, and region-based material type assignments, are both acceptable partitions (see Ref. [4] for information about this and other meta-data often accompanying mesh). In addition to defining the groupings of regions into parts, the assignment of specific parts to processors can be done implicitly or using additional data stored with the partition.

MOAB implements several specific methods for loading mesh into a parallel representation:

- READ_PART: each processor reads only the mesh used by its part(s).
- READ_DELETE: each processor reads the entire mesh, then deletes the mesh not used by its part(s).
- BCAST_DELETE: the root processor reads and broadcasts the mesh; each processor then deletes the mesh not used by its part(s).

The READ_DELETE and BCAST_DELETE methods are supported for all file types MOAB is able to read, while READ_PART is only implemented for MOAB's native HDF5-based file format.

Various other options control the selection of part sets or other details of the parallel read process. For example, the application can specify the tags, and optionally tag values, which identify parts, and whether those parts should be distributed according to tag value or using round-robin assignment.

The options used to specify loading method, the data used to identify parts, and other parameters controlling the parallel read process, are shown in Table 5.

Table 5: Options passed to MOAB's load_file function identifying the partition and other parameters controlling the parallel read of mesh data. Options and values should appear in option string as "option=val", with a delimiter (usually ";") between options.

Option	Value	Description
PARTITION	<tag_name>	Sets with the specified tag name should be used as part sets
PARTITION_VAL	<val1, val2-val3, ...>	Integer values to be combined with tag name, with ranges input using val1, val2-val3. Not meaningful unless PARTITION option is also given.
PARTITION_ DISTRIBUTE	(none)	If present, or values are not input using PARTITION_VAL, sets with tag indicated in PARTITION option are partitioned across processors in round-robin fashion.
PARALLEL_RESOLVE _SHARED_ENTS	<pd.sd>	Resolve entities shared between processors, where partition is made up of *pd*- dimensional entities, and entities of dimension *sd* and lower should be resolved.
PARALLEL_GHOSTS	<gd.bd.nl[.ad]>	Exchange ghost elements at shared inter-processor interfaces. Ghost elements of dimension *gd* will be exchanged. Ghost elements are chosen going through *bd*-dimensional interface entities. Number of layers of ghost elements is specified in *nl*. If *ad* is present, lower-dimensional entities bounding exchanged ghost entities will also be exchanged; allowed values for *ad* are 1 (exchange bounding edges), 2 (faces), or 3 (edges and faces).
CPUTIME	(none)	Print cpu time required for each step of parallel read & initialization.
MPI_IO_RANK	<r>	If read method requires reading mesh onto a single processor, processor with rank *r* is used to do that read.

Several example option strings controlling parallel reading and initialization are:

"PARALLEL=READ_DELETE; PARTITION=MATERIAL_SET; PARTITION_VAL=100, 200, 600-700": The whole mesh is read by every processor; this processor keeps mesh in sets assigned the tag whose name is "MATERIAL_SET" and whose value is any one of 100, 200, and 600-700 inclusive.

"PARALLEL=READ_PART; PARTITION=PARALLEL_PARTITION, PARTITION_VAL=2": Each processor reads only its mesh; this processor, whose rank is 2, is responsible for elements in a set with the PARALLEL_PARTITION tag whose value is 2. This would by typical input for a mesh which had already been partitioned with e.g. Zoltan or Parmetis.

"PARALLEL=BCAST_DELETE; PARTITION=GEOM_DIMENSION, PARTITION_VAL=3, PARTITION_DISTRIBUTE": The root processor reads the file and broadcasts the whole mesh to all processors. If a list is constructed with entity sets whose GEOM_DIMENSION tag is 3, i.e. sets corresponding to geometric volumes in the original geometric model, this processor is responsible for all elements with index R+iP, i >= 0 (i.e. a round-robin distribution).

5.2.2. Parallel Mesh Initialization Using Functions

After creating the local mesh on each processor, an application can call the following functions in MBParallelComm to establish information on shared mesh entities. See the [ref-directparmesh example] in the MOAB source tree for a complete example of how this is done from an application.

- **ParallelComm::resolve_shared_entities (collective):** Resolves shared entities between processors, based on GLOBAL_ID tag values of vertices. Various forms are available, based on entities to be evaluated and maximum dimension for which entity sharing should be found.
- **ParallelComm::exchange_ghost_cells (collective):** Exchange ghost entities with processors sharing an interface with this processor, based on specified ghost dimension (dimension of ghost entities exchanged), bridge dimension, number of layers, and type of adjacencies to ghost entities. An entity is sent as a ghost if it is within that number of layers, across entities of the bridge dimension, with entities owned by the receiving processor, or if it is a lower-dimensional entity adjacent to a ghost entity and that option is requested.

5.2.3. Communicator Handling

The ParallelComm constructor takes arguments for an MPI communicator and a MOAB instance. The ParallelComm instance stores the MPI communicator, and registers itself with the MOAB instance. Applications can specify the ParallelComm index to be used for a given file operation, thereby specifying the MPI communicator for that parallel operation. For example:

```
// pass a communicator to the constructor, getting back the index
MPI_Comm my_mpicomm;
int pcomm_index;
ParallelComm my_pcomm(moab, my_mpicomm, &pcomm_index);

// write the pcomm index into a string option
char load_opt[32];
sprintf(load_opt, "PARALLEL=BCAST_DELETE;PARALLEL_COMM=%d",
   pcomm_index);

// specify that option in a parallel read operation
```

```
MBErrorCode rval = moab->load_file(load_opt, fname, ...)
```

In the above code fragment, the ParallelComm instance with index pcomm_index will be used in the parallel file read, so that the operation executes over the specified MPI communicator. If no ParallelComm instance is specified for a parallel file operation, a default instance will be defined, using MPI_COMM_WORLD.

Applications needing to retrieve a ParallelComm instance created previously and stored with the MOAB instance, e.g. by a different code component, can do so using a static function on ParallelComm:

```
ParallelComm *my_pcomm = ParallelComm::get_pcomm(moab, pcomm_index);
```

ParallelComm also provides the ParallelComm::get_all_pcomm function, for retrieving all ParallelComm instances stored with a MOAB instance. For syntax and usage of this function, see the MOAB online documentation for ParallelComm.hpp [8].

5.3. Parallel Mesh Query Functions

Various functions are commonly used in parallel mesh-based applications. Functions marked as being collective must be called collectively for all processors that are members of the communicator associated with the ParallelComm instance used for the call.

ParallelComm::get_pstatus: Get the parallel status for the entity.

ParallelComm::get_pstatus_entities: Get all entities whose pstatus satisfies (pstatus & val).

ParallelComm::get_owner: Get the rank of the owning processor for the specified entity.

ParallelComm::get_owner_handle: Get the rank of the owning processor for the specified entity, and the entity's handle on the owning processor.

ParallelComm::get_sharing_data: Get the sharing processor(s) and handle(s) for an entity or entities. Various overloaded versions are available, some with an optional "operation" argument, where Interface::INTERSECT or Interface::UNION can be specified. This is similar to the operation arguments to Interface::get_adjacencies.

ParallelComm::get_shared_entities: Get entities shared with the given processor, or with all processors. This function has optional arguments for specifying dimension, whether interface entities are requested, and whether to return just owned entities.

ParallelComm::get_interface_procs: Return all processors with whom this processor shares an interface.

ParallelComm::get_comm_procs: Return all processors with whom this processor communicates.

5.4. Parallel Mesh Communication

Once a parallel mesh has been initialized, applications can call the ParallelComm::exchange_tags function for exchanging tag values between processors. This function causes the owning processor to send the specified tag values for all shared, owned entities to other processors sharing those entities. Asynchronous communication is used to hide latency, and only point-to-point communication is used in these calls.

6. Building MOAB-Based Applications

There are two primary mechanisms supported by MOAB for building applications, one based on MOAB-defined make variables, and the other based on the use of libtool and autoconf. Both assume the use of a "make"-based build system.

The easiest way to incorporate MOAB into an application's build process is to include the "moab.make" file into the application's Makefile, adding the make variables MOAB_INCLUDES and MOAB_LIBS_LINK to application's compile and link commands, respectively. MOAB_INCLUDES contains compiler options specifying the location of MOAB include files, and any preprocessor definitions required by MOAB. MOAB_LIBS_LINK contains both the options telling where libraries can be found, and the link options which incorporate those libraries into the application. Any libraries depended on by the particular configuration of MOAB are included in that definition, e.g. the HDF5 library. Using this method to incorporate MOAB is the

most straightforward; for example, the following Makefile is used to build one of the example problems packaged with the MOAB source:

```
include ${MOAB_LIB_DIR}/moab.make

GetEntities : GetEntities.o
   ${CXX} $< ${MOAB_LIBS_LINK} -o $@

.cpp.o :
   ${CXX} ${MOAB_INCLUDES} -c $<
```

Here, the MOAB_LIB_DIR environment variable or make argument definition specifies where the MOAB library is installed; this is also the location of the moab.make file. Once that file has been included, MOAB_INCLUDES and MOAB_LIBS_LINK can be used, as shown.

Other make variables are defined in the moab.make file which simplify building applications:

- MOAB_LIBDIR, MOAB_INCLUDEDIR: the directories into which MOAB libraries and include files will be installed, respectively. Note that some include files are put in a subdirectory named "moab" below that directory, to reflect namespace naming conventions used in MOAB.
- MOAB_CXXFLAGS, MOAB_CFLAGS, MOAB_LDFLAGS: Options passed to the C++ and C compilers and the linker, respectively.
- MOAB_CXX, MOAB_CC, MOAB_FC: C++, C, and Fortran compilers specified to MOAB at configure time, respectively.

The second method for incorporating MOAB into an application's build system is to use autoconf and libtool. MOAB is configured using these tools, and generates the ".la" files that hold information on library dependencies that can be used in application build systems also based on autoconf and libtool. Further information on this subject is beyond the scope of this User's Guide; see the ".la" files as installed by MOAB, and contact the MOAB developer's mailing list [6] for more details.

7. iMesh (ITAPS Mesh Interface) Implementation in MOAB

iMesh is a common API to mesh data developed as part of the Interoperable Tools for Advanced Petascale Simulations (ITAPS) project [18]. Applications using the iMesh interface can operate on any implementation of that interface, including MOAB. MOAB-based applications can take advantage of other services implemented on top of iMesh, including the MESQUITE mesh improvement toolkit [19] and the GRUMMP mesh improvement library [20].

MOAB's native interface is accessed through the MBInterface abstract C++ base class. Wrappers are not provided in other languages; rather, applications wanting to access MOAB from those languages should do so through iMesh. In most cases, the data models and functionality available through MOAB and iMesh are identical. However, there are a few differences, subtle and not-so-subtle, between the two:

SPARSE tags used by default: MOAB's iMesh implementation creates SPARSE tags by default, because of semantic requirements of other tag-related functions in iMesh.

Higher-order elements: ITAPS currently handles higher-order elements (e.g. a 10-node tetrahedron) usi[21][4]. As described in [sec-entities], MOAB supports higher-order entities by allowing various numbers of vertices to define topological entities like quadrilateral or tetrahedron. Applications can specify flags to the connectivity and adjacency functions specifying whether corner or all vertices are requested.

Self-adjacencies: In MOAB's native interface, entities are always self-adjacent[5]; that is, adjacencies of equal dimension requested from an entity will always include that entity, while from iMesh will not include that entity.

[4] There are currently no implementations of this interface.

[5] iMesh and MOAB both define adjacencies using the topological concept of closure. Since the closure of an entity includes the entity itself, the d-dimensional entities on the closure of a given entity should include the entity itself.

To provide complete MOAB support from other languages through iMesh, a collection of iMesh extension functions are also available. A general description of these extensions appears below; for a complete description, see the online documentation for iMesh-extensions.h [8].

- **Recursive get_entities functions:** There are many cases where sets include other sets (see [4] for more information). MOAB provides iMesh_getEntitiesRec, and other recursive-supporting functions, to get all non-set entities of a given type or topology accessible from input set(s). Similar functions are available for number of entities of a given type/topology.
- **Get entities by tag, and optionally tag value:** It is common to search for entities with a given tag, and possibly tag value(s); functions like iMesh_getEntitiesByTag are provided for this purpose.
- **Options to createTag:** To provide more control over the tag type, the iMesh_createTagByOption is provided.
- **MBCNType:** Canonical numbering evaluations are commonly needed by applications, e.g. to apply boundary conditions locally. The MBCN package provides these evaluations in terms of entity types defined in MOAB [9]; the getMBCNType is required to translate between iMesh_Topology and MBCN type.

As required by the iMesh specification, MOAB generates the "iMesh-Defs.inc" file and installs it with the iMesh and MOAB libraries. This file defines make variables which can be used to build iMesh-based applications. The method used here is quite similar to that used for MOAB itself (see Section 6). In particular, the IMESH_INCLUDES and IMESH_LIBS make variables can be used with application compile and link commands, respectively, with other make variables similar to those provided in moab.make also available.

Note that using the iMesh interface from Fortran-based applications requires a compiler that supports Cray pointers, along with the pass-by-value (%VAL) extension. Almost all compilers support those extensions; however, if using the gcc series of compilers, you must use gfortran 4.3 or later.

8. Structured Mesh Representation

A structured mesh is defined as a D-dimensional mesh whose interior vertices have 2D connected edges. Structured mesh can be stored without connectivity, if certain information is kept about the parametric space of each structured block of mesh. MOAB can represent structured mesh with implicit connectivity, saving approximately 57% of the storage cost compared to an unstructured representation[6]. Since connectivity must be computed on the fly, these queries execute a bit slower than those for unstructured mesh. More information on the theory behind MOAB's structured mesh representation can be found in

Currently, MOAB's structured mesh representation can only be used by creating structured mesh at runtime; that is, structured mesh is saved/restored in an unstructured format in MOAB's HDF5-based native save format. For more details on how to use MOAB's structured mesh representation, see the scdseq_test.cpp source file in the test/ directory.

9. Performance and Using MOAB Efficiently from Applications

MOAB is designed to operate efficiently on groups of entities and for large meshes. Applications will be most efficient when they operate on entities in groups, especially groups which are close in their order of creation. The MOAB API is structured to encourage operations on groups of entities. Conversely, MOAB will not perform as well as other libraries if there are frequent deletion and creation of entities. For those types of applications, a mesh library using a C++ object-based representation is more appropriate. In this section, performance of MOAB when executing a variety of tasks is described, and compared to that of other representations. Of course, these metrics are based on the particular models and environments where they are run, and may or may not be representative of other application types.

[6]This assumes vertex coordinates are represented explicitly, and that there are approximately the same number of vertices and hexahedra in a structured hex mesh.

One useful measure of MOAB performance is in the representation and query of a large mesh. MOAB includes a performance test, located in the test/perf directory, in which a single rectangular region of hexahedral elements is created then queried; the following steps are performed:

- Create the vertices and hexes in the mesh
- For each vertex, get the set of connected hexahedra
- For each hex, get the connected vertices, their coordinates, average them, and assign them as a tag on the hexes

This test can be run on your system to determine the runtime and memory performance for these queries in MOAB.

10. Conclusions and Future Plans

MOAB, a Mesh-Oriented datABase, provides a simple but powerful data abstraction to structured and unstructured mesh, and makes that abstraction available through a function API. MOAB provides the mesh representation for the VERDE mesh verification tool, which demonstrates some of the powerful mesh metadata representation capabilities in MOAB. MOAB includes modules that import mesh in the ExodusII, CUBIT .cub and Vtk file formats, as well as the capability to write mesh to ExodusII, all without licensing restrictions normally found in ExodusII-based applications. MOAB also has the capability to represent and query structured mesh in a way that optimizes storage space using the parametric space of a structured mesh; see Ref. [17] for details.

Initial results have demonstrated that the data abstraction provided by MOAB is powerful enough to represent many different kinds of mesh data found in real applications, including geometric topology groupings and relations, boundary condition groupings, and inter-processor interface representation. Our future plans are to further explore how these abstractions can be used in the design through analysis process.

11. References

[1] M. Fatenad and G.A. Moses, "Cooper radiation hydrodynamics code.."

[2] T.J. Tautges and A. Caceres, "Scalable parallel solution coupling for multiphysics reactor simulation," *Journal of Physics: Conference Series*, vol. 180, 2009.

[3] T.J. Tautges, *MOAB Meta-Data Information*, 2010.

[4] T.J. Tautges, "MOAB - ITAPS – Trac.", http://trac.mcs.anl.gov/projects/ITAPS/wiki/MOAB

[5] "MOAB Developers Email List.", moab-dev@mcs.anl.gov.

[6] "MOAB Users Email List.", moab@mcs.anl.gov.

[7] "MOAB online documentation.", http://gnep.mcs.anl.gov:8010/moab-docs/

[8] T.J. Tautges, "Canonical numbering systems for finite-element codes," *Communications in Numerical Methods in Engineering*, vol. Online, Mar. 2009.

[9] L.A. Schoof and V.R. Yarberry, *EXODUS II: A Finite Element Data Model*, Albuquerque, NM: Sandia National Laboratories, 1994.

[10] M. PATRAN, "PATRAN User's Manual," 2005.

[11] *VisIt User's Guide*.

[12] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan, "Zoltan Data Management Services for Parallel Dynamic Applications," *Computing in Science and Engineering*, vol. 4, 2002, pp. 90–97.

[13] T.J. Tautges, P.P.H. Wilson, J. Kraftcheck, B.F. Smith, and D.L. Henderson, "Acceleration Techniques for Direct Use of CAD-Based Geometries in Monte Carlo Radiation Transport," *International Conference on Mathematics, Computational Methods & Reactor Physics (M&C 2009)*, Saratoga Springs, NY: American Nuclear Society, 2009.

[14] H. Kim and T. Tautges, "EBMesh: An Embedded Boundary Meshing Tool," *in preparation*.

[15] G.D. Sjaardema, T.J. Tautges, T.J. Wilson, S.J. Owen, T.D. Blacker, W.J. Bohnhoff, T.L. Edwards, J.R. Hipp, R.R. Lober, and S.A. Mitchell, *CUBIT mesh generation environment Volume 1: Users manual*, Sandia National Laboratories, May 1994, 1994.

[16] T.J. Tautges, "CGM: A geometry interface for mesh generation, analysis and other applications," *Engineering with Computers*, vol. 17, 2001, pp. 299-314.

[17] T. J. Tautges, MOAB-SD: Integrated structured and unstructured mesh representation, *Engineering with Computers*,

vol. 20, no. 3, pp. 286-293, 2004.
[18] "Interoperable Technologies for Advanced Petascale Simulations (ITAPS)," *Interoperable Technologies for Advanced Petascale Simulations (ITAPS)*.
[19] P. Knupp, "Mesh quality improvement for SciDAC applications," *Journal of Physics: Conference Series*, vol. 46, 2006, pp. 458-462.

A. Building & Installing

MOAB uses an autoconf and libtool-based build process by default. The procedure used to build MOAB from scratch depends on whether the source code was obtained from a "tarball" or directly from the Subversion repository. Assuming the latter, the following steps should be executed for building and installing MOAB:

1. Locate and build any required dependencies. MOAB can be built with no dependencies on other libraries; this may be useful for applications only needing basic mesh representation and not needing to export mesh to formats implemented in other libraries. MOAB's native save/restore capability is built on HDF5-based files; applications needing to save and restore files from MOAB reliably should use this library. MOAB also uses ExodusII, a netCDF-based file format developed at Sandia National Laboratories [10]. Applications needing to execute these tests should also build netCDF. Note that MOAB uses netCDF's C++ interface, which is not enabled by default in netCDF but can be enabled using the "–enable-cxx" option to netCDF's configure script.
2. Unpack source code into <moab>, and change current working directory to that location.
3. Execute "autoreconf –fi".
4. Run configure script, by executing "./configure <options>". Recommended options:
    a. –prefix=<install_dir>: directory below which MOAB library and include files will be installed; can either be the directory used for MOAB source (<moab> from step 1), or a different directory.
    b. –hdf5-dir=<hdf5_dir>: directory whose "include" and "lib" subdirectories hold HDF5 include and library, respectively. MOAB uses HDF5 for its native save/restore format (see Section 4.6.1).
    c. –netcdf-dir=: directory whose "include" and "lib" subdirectories hold netCDF include and library, respectively. MOAB uses netCDF-based files for many of its build tests. If the location of netCDF cannot be found, MOAB's build tests will not function properly, but MOAB will still be usable.
5. Run "make check"; this runs a series of build tests, to verify that the MOAB build was successful. Note this check will fail if netCDF is not used, but MOAB itself will still be usable from applications.
6. Run "make install"; this copies include files and libraries to subdirectories of the directory specified in the "prefix" option.

These steps are sufficient for building MOAB against HDF5 and netCDF. By default, a small number of standard MOAB-based applications are also built, including mbconvert (a utility for reading and writing files), mbsize (for querying basic information about a mesh), and the iMesh interface (see Section 7). Other utilities can be enabled using various other options to the configure script; for a complete list of build options, execute "./configure –help".

B. Differences Between iMesh and MOAB

The data models used in MOAB and iMesh are quite similar, but not identical. The most significant differences are the following:

- Tags: MOAB differentiates between DENSE, SPARSE, and BIT tags, using different storage models for each, while iMesh uses a single tag concept. iMesh allows application to query whether an entity has been given a tag of a specified type; this query is incompatible with the concept of a DENSE tag with a

default value. Thus, MOAB's iMesh implementation creates SPARSE tags by default, and tags created and accessed through this interface will use more memory than DENSE tags created through MOAB's native interface. To mitigate this problem, MOAB implements an extension of the iMesh_createTag function which allows specification of the tag type (DENSE, SPARSE, etc.) to be created. See later in this section for more information.

- Higher-order nodes: ITAPS currently handles higher-order elements (e.g. a 10-node tetrahedron) using a special "Shape" interface. In this interface, higher-order nodes are only accessible through the AEntities which they resolve. MOAB's iMesh implementation provides access to higher-order nodes in the same manner described in Section , by varying the number of vertices defining each entity. As a result, if higher-order entities are used in a model, the functions returning connectivity and vertex adjacencies always return all vertices, rather than providing an option to return just corner vertices.
- Self-adjacencies: iMesh specifies that entities are not self-adjacent; that is, requesting adjacencies of the same dimension/type results in an error. MOAB does not consider this an error, returning the entity itself.
- Adjacency table and AEntities: iMesh uses the concept of an "adjacency table" to determine which AEntities are available and created by default. MOAB uses input arguments to the get_adjacencies functions to control whether AEntities are created. These flags provide finer-grained control over AEntities, but make it slightly less convenient to ensure that AEntities of a given dimension are always created.

X. MOAB Parallel Mesh Representation

At a high level, access to the mesh on both serial and parallel machines is done through the local MOAB interface object. Thus, on a parallel machine, most mesh data is accessed on each processor as if it was a serial mesh. A communicator-like class, ParallelComm, provides convenience functions for accessing parallel constructs in the mesh directly. However, parallel aspects of the mesh are represented using the standard MOAB data model, using a combination of sets and tags, and can be accessed that way in some cases[7]. The following constructs are used to represent the various parallel aspects of mesh data; the tags used to store parallel mesh constructs are summarized in Table 6.

Table 6: Tags used to store parallel mesh information in MOAB's data model. Note that tags whose name begins with double-underscore ("__") are not saved to disk when a mesh is written. NP is the MAX_SHARING_PROCS preprocessor variable defined in ParallelComm.hpp.

Tag name	Data type	Applies to (E=entity, S=set)	Purpose
PARALLEL_PART	I	S	Represent a part in a partition
PARALLEL_ PARTITION	I	S	Represents a partition of the mesh for parallel solution, which is a collection of parts
__PARALLEL_SHARED_ PROC	I	E, S	Rank of other processor sharing this entity/set
__PARALLEL_SHARED_ HANDLE	H	E, S	Handle of this entity/set on sharing processor
__PARALLEL_SHARED_ PROCS	I*NP	E, S	Ranks of other processors sharing this entity/set
__PARALLEL_SHARED_ HANDLES	H*NP	E, S	Handles of this entity/set on sharing processors

[7]Note, the specific tags and sets used for storing parallel mesh information may change in the future, thus it is safer to use the convenience functions in ParallelComm for accessing this information.

__PARALLEL_STATUS	C*1	E, S	Bit-field indicating various parallel information

Shared entities: Entities and interface sets shared with a *single* other processor are referred to as "shared", with multiple other processors as *multi-shared*. Shared entities are given the tags "__PARALLEL_SHARED_PROC" and "__PARALLEL_SHARED_HANDLE" tags, which store the remote processor and the handle of the entity on that processor, respectively. These tags are implemented as "dense" tags, with the default value of -1 and 0 for processor/handle. Multi-shared entities and sets are given the "__PARALLEL_SHARED_PROCS" and "__PARALLEL_SHARED_HANDLES" tags. These tags are sized large enough to store the maximum number of processors likely to share a given entity; this value is controlled by the MAX_SHARING_PROCS preprocessor variable in ParallelComm.hpp, and has a default value of 64. Since these tags are a fixed size, they must be padded beyond the number of sharing processors with the corresponding invalid values for those fields (-1 for processor, 0 for handle). Note that the local processor rank and the entity handle on that processor are stored in the __PARALELL_SHARED_PROCS/HANDLES tags.

Parallel status: A characteristic of entities and sets represented in parallel. The parallel status for both entities and sets is only relevant when running in parallel, and is represented with a character-size dense tag __PSTATUS. Bits in this tag, and the parallel characteristics they represent, are shown in Table 4.

Table 7: Bits in the __PSTATUS tag representing various parallel characteristics of a mesh. Also listed are enumerated values that can be used in bitmask expressions.

Bit	Name	Represents
0x1	PSTATUS_NOT_OWNED	Not owned by the local processor
0x2	PSTATUS_SHARED	Shared by exactly two processors
0x4	PSTATUS_MULTISHARED	Shared by three or more processors
0x8	PSTATUS_INTERFACE	Part of lower-dimensional interface shared by multiple processors
0x10	PSTATUS_GHOST	Represented locally by not owned and not interface

Owning Processor: Each entity is owned by a single processor. The processor owning an entity is indicated using a combination of tags. If the PSTATUS_NOT_OWNED bit in an entity's PSTATUS tag is not set, the entity is owned locally, and no further information is needed. If this bit is set, then the processor owning the entity is indicated either by value of the __PARALLEL_SHARED_PROC tag (shared entities) or the first value in the __PARALLEL_SHARED_PROCS tag (multi-shared entities).

Inter-Processor Interface: A collection of lower-dimensional entities adjacent to higher-dimensional entities in two or more parts; can be either shared or multi-shared. Interface sets are marked as such in their PSTATUS tags, and are given the PARALLEL_SHARED_... tags to indicate sharing processors/handles.

Part: A collection of entities, usually regions, assigned to a single processor. Parts are represented by entity sets, marked with the PARALLEL_PART tag whose value indicates the rank of the processor to which that part is assigned.

Partition: A collection of parts, partitioning a collection of entities such that each entity is contained in exactly one part; there can be more than one partition stored with a given mesh, or none. A partition can be computed, stored, and accessed on a mesh in serial, and any collection of sets with the characteristics described above can be considered a partition. Partitions are represented by their own entity set, and are marked with the PARALLEL_PARTITION tag.